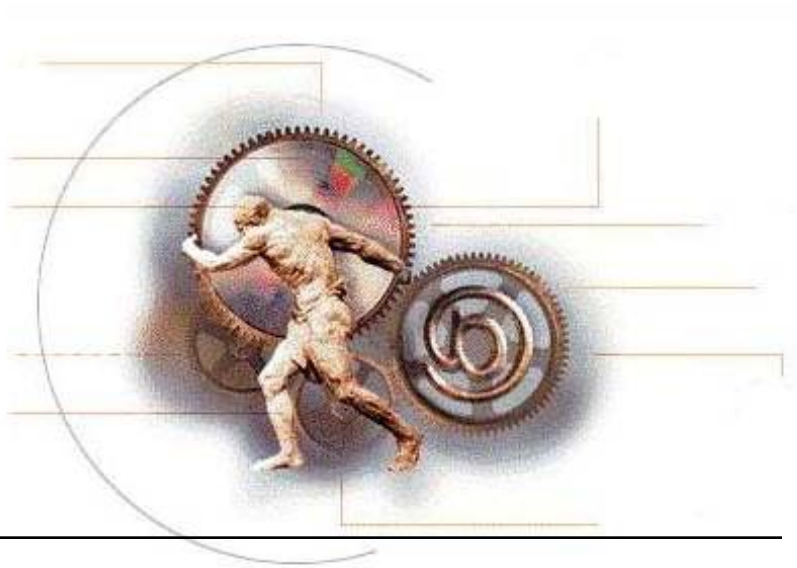


Course 7

貪婪法則
Greedy Approach

國立聯合大學 資訊管理學系 陳士杰老師

Outlines

- 本章重點
  - Dynamic Programming v.s. Greedy Approach
  - Concepts of Greedy Approach
  - Minimum Spanning Trees
  - The Greedy Approach versus Dynamic Programming: The Knapsack Problem
  - Dijkstra Algorithm for Single-pair Shortest Path Problem

Dynamic Programming v.s. Greedy Approach

- 對於具有限制的最佳化問題，可以採用 "貪婪法則" 或 "動態規劃" 來設計演算法則。
  - 所謂具有限制條件的最佳化問題，是指可以將這一個問題表示成為具有一個目標函數 (Objective Function)與一些限制函數 (Constraint Function；或稱限制條件)的式子。
  - 對於求解具有限制條件的最佳化問題時所得到的不同答案類型而言:
    - 符合限制函數 (條件) 的所有答案，一般通稱為可行解 (Feasible Solution)
    - 但是在這一群可行解中，如果能夠讓目標函數最佳化，則這一個可行解就稱為最佳解 (Optimal Solution)

- **Greedy Approach**
  - 是一種階段性 (Stage) 的方法
  - 具有一選擇程序 (Selection Procedure)，自某起始點(值) 開始，在每一個階段逐一檢查每一個輸入是否適合加入答案中，重複經過多個階段後，即可順利獲得最佳解
    - 一個選擇程序正確與否，會影響貪婪法則所設計出之演算法在執行過後的答案是否為最佳答案。
  - 較為簡單
  - 如果所要處理的最佳化問題無法找到一個選擇程序，則需要考慮所有的可能情況，就是屬於Dynamic Programming
- **Dynamic Programming**
  - 先把所有的情況都看過一遍，才去挑出最佳的結果
  - 考慮問題所有可能的情況，將最佳化問題的目標函數表示成一個遞迴關係式，結合Table的使用以找出最佳解

Concepts of Greedy Approach

- Greedy approach 從一組資料序列中抓取資料時，每一階段要抓一個該階段最佳的資料是根據一些準則 (選擇程序) 來決定，且此次的決定和先前及往後的階段所做之任何一個決定無關。
- 設計方法：
  - 根據問題的目標函數，找出一個選擇程序 (Selection Procedure)。
  - 根據這一個選擇程序，由所有的輸入中，每次逐一選擇一個最佳的輸入加以檢查，如果這一個輸入可以符合問題的限制條件，則將這一個輸入加入；反之則必須捨棄這一個輸入。
  - 每一階段可以重覆上述選擇、檢查的程序。所有階段執行完畢後，最後可以得到一個最佳解或是不存在任何一組可行解。

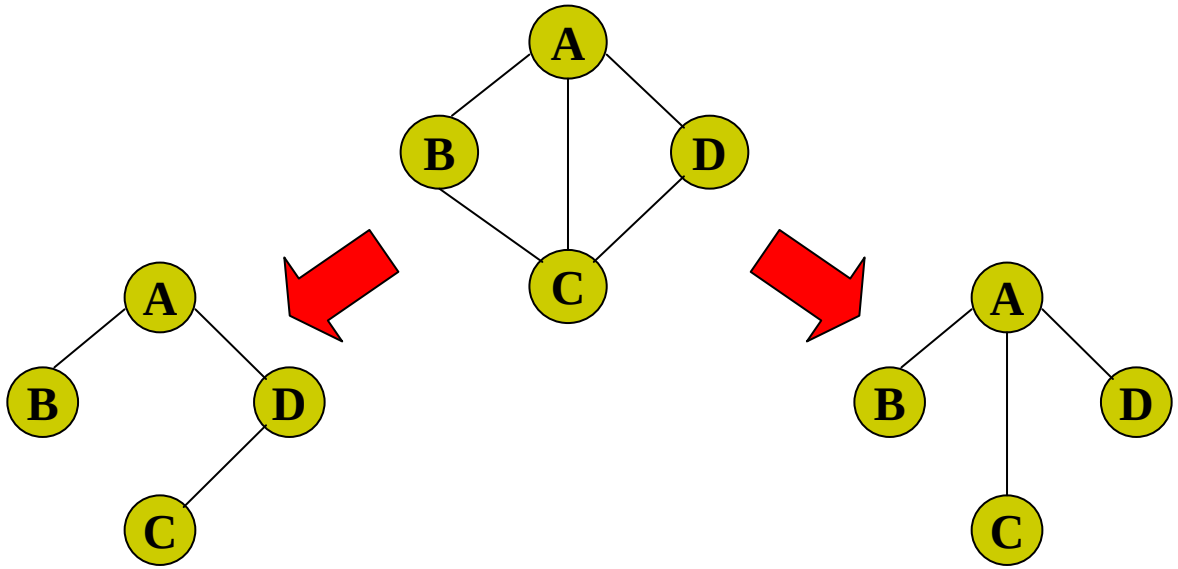
- 貪婪演算法的演算過程由一個空的解集合開始，藉由循序的加入新的解直到符合問題需求的最終解得到為止。
- 重複下列程序:
  - 選擇程序 (Selection procedure)
    - 挑選出下一個項目加入到解集合中。
    - 選擇程序的執行，是根據滿足每一階段之局部最佳化條件(locally optimal consideration)的貪婪原則來進行。
  - 可行性檢查 (Feasibility check)
    - 決定加入新項目後的新的解集合，是否在可行解的範圍之內。
  - 解答檢查 (Solution check)
    - 決定此新的解集合是否為此問題的最終結果。

- **[找零錢問題]:** 售貨員在找零錢問題中，不但要找對錢 (限制條件)，而且還要找給顧客最少的銅板 (目標函數)。
- 利用Greedy Approach如下 **(例：要找給客人75元)**：
  - **選擇程序 (selection procedure):**
    - 售貨員開始找尋收銀機中**最大幣值的硬幣**時，選擇的準則是究竟哪一枚硬幣的幣值是目前最佳的選擇 (局部最佳解)
  - **可行性檢查 (feasibility check):**
    - 售貨員必須判斷他剛剛選擇出那一枚硬幣的幣值加上 "目前顧客方已經收到的幣值總數" 是否超過 "應找給顧客的最後總數"。(是否有超過75元)
  - **解答檢查 (solution check):**
    - 售貨員必須檢查目前 "已找給顧客方的零錢總數" 是否等於 "應找給顧客的最後總數"。(是否已經等於75元)
    - 如果兩者不相等，則售貨員必須繼續利用他的選擇硬幣機制拿出硬幣，並重複上述三個過程直到 **"已找給顧客方的零錢總數"** 等於 **"應找給顧客的最後總數"**; 或是收銀機裡的硬幣全部用盡為止。

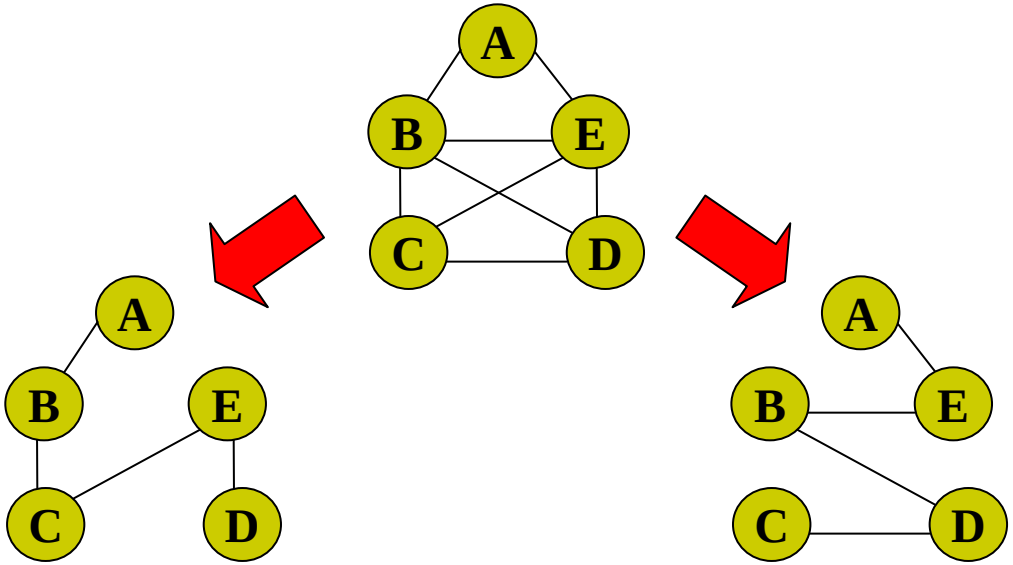
```
while ( there are more coins and the instance is not solved){
  grab the largest remaining coin;                    // selection procedure
  If (adding the coin makes the change exceed the
                  amount owed)                        // feasibility check
     reject the coin;
  else
     add the coin to the change;
  If (the total value of the change equals the
                  amount owed)                        // solution check
      the instance is solved;
}
```

Minimum Spanning Trees (最小成本擴張樹)

- Spanning Tree (擴張樹)
  - Def: G = <V, E>為一Connected無向圖，令F為追蹤Graph時所經過的邊集合，B為未經過的邊集合，則S = (V, F)為G的一個Spanning Tree，且S滿足:
    - E = F+B
    - 自B中任取一邊加入S中，必形成Cycle
    - 在S中，任何頂點對之間必存在一唯一Simple path。
- Ex:



- [Note]:
  - S中的V等於G中的V
  - 若G 不連通 (**not connected**)，則G無Spanning Tree
  - G中的Spanning Tree不只一個
  - 若|V|=n，則|F| = n-1
  - 同一G中的任二個不同之Spanning Tree不一定有交集的邊存在
    - Ex:



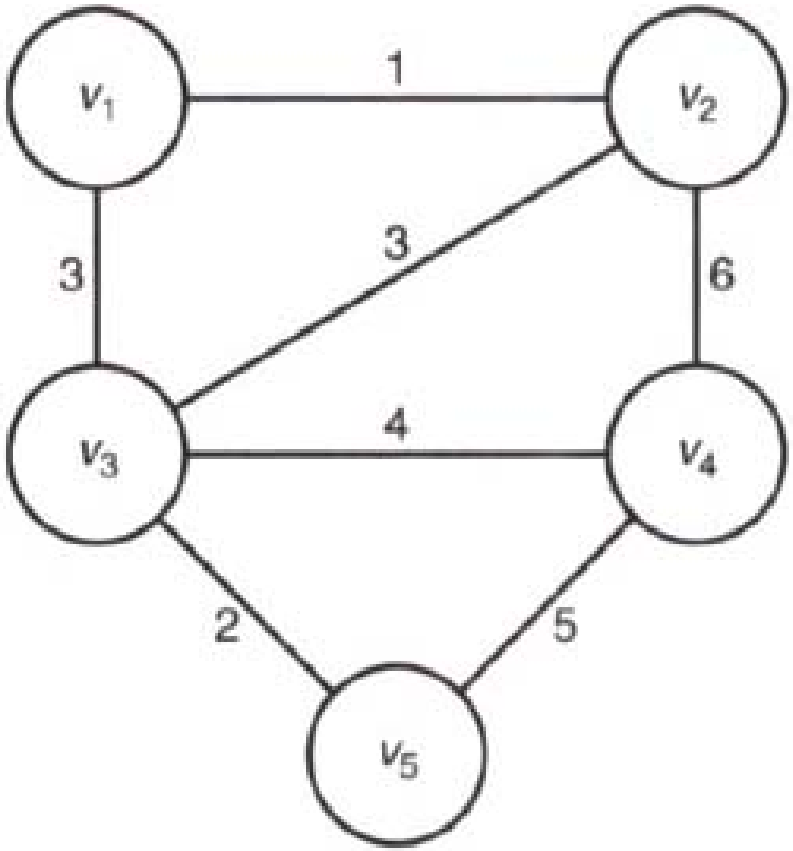
Minimal Spanning Tree (最小成本擴張樹)

- **Def:**
  - 給予一個**Connected** 的無向圖 **G=(V, E)**，且邊上具有成本 **(Cost)**或加權值 **(Weight)**，則在**G** 的所有**Spanning Tree**中，具有**最小的邊成本 (加權) 總和者**稱之。
- 應用:
  - 電路佈局的最小成本
  - 連接**n** 個城市之交通連線之最少架設成本
  - 旅遊**n**個城市之最少花費 (不回原點)
- **Algorithm:**
  - **Kruskal's Algorithm**
  - **Prim's Algorithm**

Kruskal's Algorithm

- **Kruskal's Algorithm 解題要件**：
  - 選擇程序：由擴張樹的所有邊中，挑選出具最小值者。
  - 限制條件：不允許有迴路
- **Steps:**
  1. 先將各邊依權重由小到大排序
  2. 建構一個空的邊集合F
  3. 自原無向圖G = (V, E)的邊集合E中，挑選出最小成本的邊，並將之從邊集合E中刪除 (其中，頂點V的個數 |V| = n)
  4. 若該邊加入Spanning Tree中未形成Cycle，則加入F中；否則放棄
  5. Repeat ③ ~ ④直到下列任一條件成立為止:
     - (n-1)個邊已挑出 //n是頂點的個數
     - 無邊可挑
  6. Check若 |F| < n-1，則無Spanning Tree

- 試利用Kruskal's Algo.求下圖的Minimum Spanning Tree

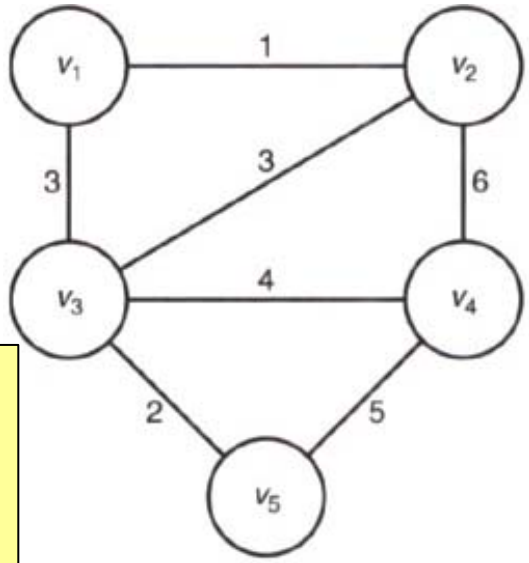


Sol:

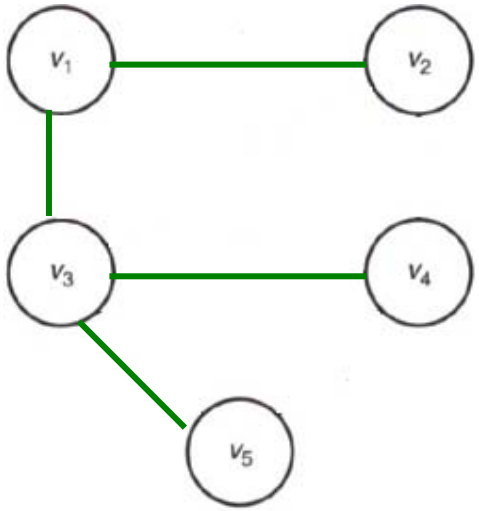
根據權重值來排列邊線

~~(V_1, V_2)~~: 1
~~(V_3, V_5)~~: 2
~~(V_1, V_3)~~: 3
~~(V_2, V_3)~~: 4
~~(V_3, V_4)~~: 5
~~(V_4, V_5)~~: 6
~~(V_2, V_4)~~: 7

該數值指的是此邊線依權重所排列的順序，不是權重!!



Set F



Min. Cost = 1+2+3+4 = 10

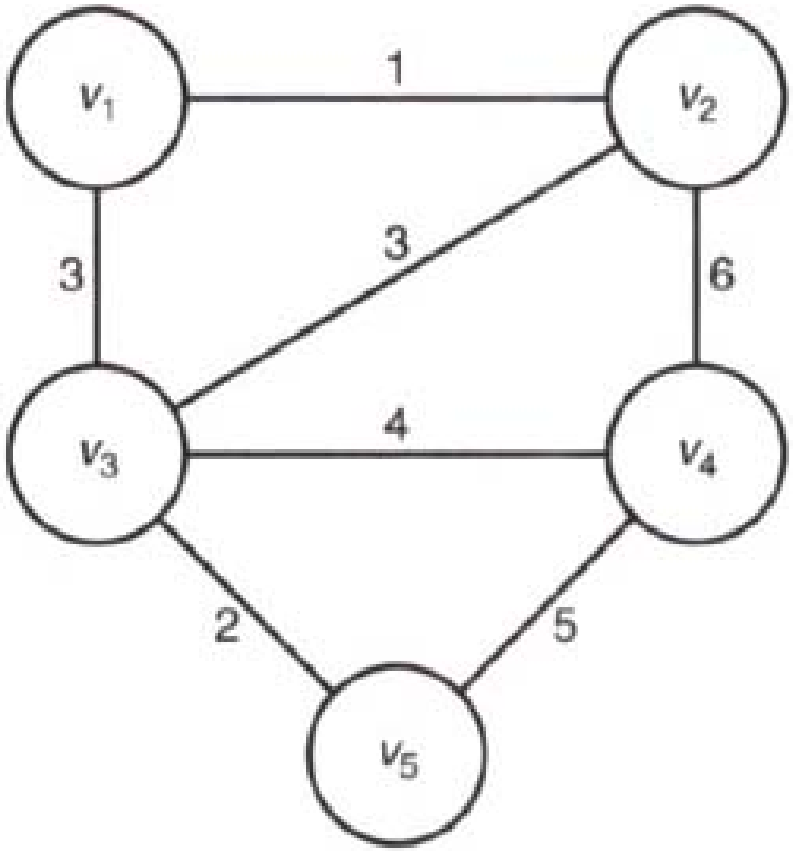
```
F = ∅                                                    // Initialize set of
                                                         // edges to empty.
create disjoint subsets of V, one for each
vertex and containing only that vertex;

sort the edges in E in nondecreasing order;
while (the instance is not solved){

  select next edge;                                      // selection procedure
  if (the edge connects two vertices in                  // feasibility check
                      disjoint subsets){
      merge the subsets;
      add the edge to F;
  }
  if ( all the subsets are merged)                       // solution check
     the instance is solved;
}
```

- 分析:
  - 先將各邊依權重由小到大排序
    - 利用 Quick Sort，當 |E| = n時 ⇨ O(n log n)
  - While迴圈
    - 當 |E| = n時 ⇨此迴圈共花費**O(n)**的時間執行:
      - 挑選最小權重的邊之工作 ⇨ O(1)
      - 判斷有無Cycle之工作 ⇨ O(1)
- 結合上面兩個工作的時間複雜度，此演算法總共花 O(n log n) + O(n) 執行工作
  - ∴ Time complexity = O(n log n)

Prim's Algorithm

- **Prims's Algorithm 解題要件**：
  - 選擇程序：由擴張樹的某一頂點與其它頂點的所有邊中，挑選出具最小值者。
  - 限制條件：不允許有迴路
- **Steps:**
  1. 建構一個空的邊集合**F**
  2. 設定兩個頂點集合及其初始值:
     - **Y = {1}** //起始頂點可任選
     - **V-Y**
  3. 挑選出具最小成本的邊 **(u, w)**，其中 **u ∈ Y**，**w ∈ V-Y**
  4. **(u, w)**自**E**中刪除，加入構成**Spanning Tree** 的邊集合**F**中，同時從**V-Y**集合中刪除**w**，並將**w**加入集合**Y**中
  5. **Repeat** ③~④直到下列任一條件成立為止:
     - **Y = V**
     - 無邊可挑
  6. 若 **|F| < n-1**，則無**Spanning Tree**

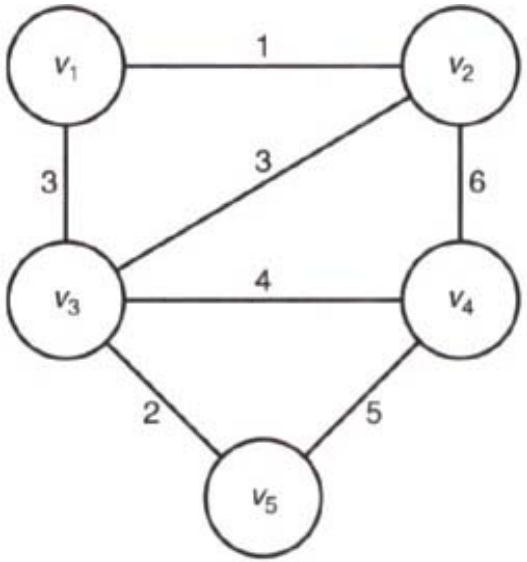
- 試利用Prim's Algo.求下圖的Minimum Spanning Tree



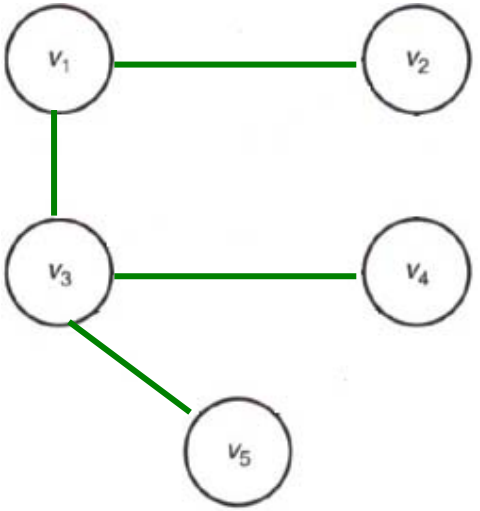
Sol:

Y	V-Y
{1}	{2, 3, 4, 5}

∵Y = V, ∴ Stop



Set F



```
F = ∅                                          // Initialize set of edges
                                               // to empty.
Y = {v1}                                          // Initialize set of vertices to
                                               // contain only the first one.
while (the instance is not solved){

  select a vertex in V - Y that is    // selection procedure and
  nearest to Y                        // feasibility check

  add the vertex to Y;
  add the edge to F;

  if (Y == V)                         // solution check
      the instance is solved;
}
```

Time Complexity: $O(n^2)$

Summary

- 這兩個演算法皆屬於 "Greedy" 策略
- 這兩個演算法以Kruskal's algo. 較為快速 (∵ n log n)
- 連通的無向圖G，其min. spanning tree不一定唯一
  - ∵ 可能會有 2 個或 2 個以上的邊具有相同的cost
- 若一連通無向圖G，其各邊cost皆不相同，則會具有唯一的min. spanning tree
- 在min. spanning tree中，各頂點之間距離並非是shortest path
  - 所有成本最小，並非其中一邊最小
  - Ex: 之前兩個演算法所用的例子，其中頂點對 (V_2, V_4) 的最短距離應為6，但所導出的兩個MST，於該頂點對的距離皆大於6!!

The Knapsack Problem (背包問題)

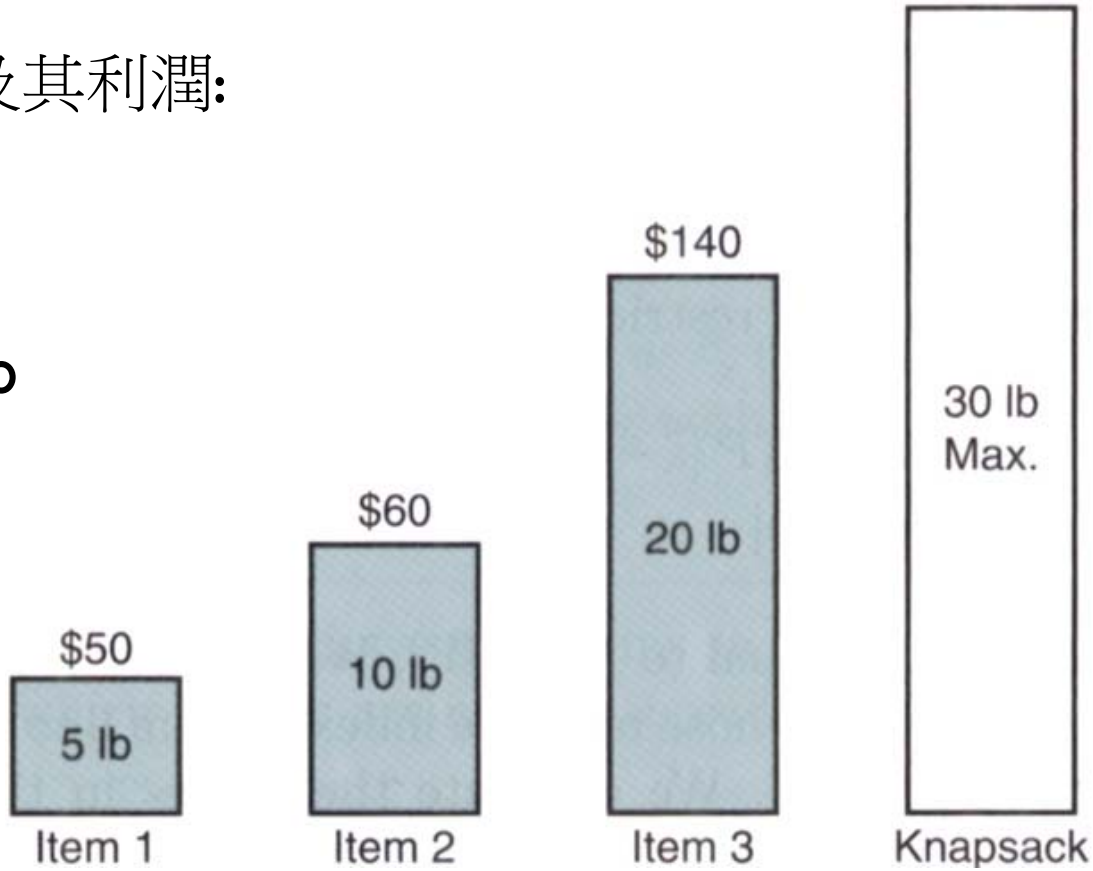
- **Def:** 所謂Knapsack Problem，是指有n個物品和一個背包，其中:
  - 物品具有重量 $(w_1, w_2, \ldots, w_n)$ 和利潤 $(p_1, p_2, \ldots, p_n)$
  - 背包的最大重量承受限制為W

  問如何取物可得最高價值?
- 此問題可以表示如下:

$$\sum_{item_i \in A} p_i \quad \text{is maximized subject to} \quad \sum_{item_i \in A} w_i \leq W.$$

- Knapsack Problem可分成兩種問題型態:
  - **Fractional Knapsack Problem**:
    - 物品可被切割，亦即取物時可取部份
    - 採用Greedy Approach
  - **0/1 Knapsack Problem**:
    - 物品不可被切割，亦即取物時得取全部
    - 採用Dynamic Programming

- 我們將以下列範例說明上述兩種類型的背包問題:
  - 背包可承擔的最大重量: 30 lb(磅)
  - 三個物品之重量及其利潤:
    - **Item 1**: 5 lb, $50
    - **Item 2**: 10 lb, $60
    - **Item 3**: 20 lb, $140



Fractional Knapsack Problem

- 物品可被切割，亦即取物時可取部份
- 採用Greedy Approach，因此需設定「選擇程序」。
  - 由於物品放入背包可以獲得利潤，但是也同時會增加重量，所以共有三種可供使用的選擇程序，分別是:
    - 利潤: 採用最大利潤優先的選擇程序。自利潤最大之物品依序取物，直到物品拿完或負重 = W為止，就可以得到一個可行解
    - 重量: 採用最小重量優先的選擇程序。自重量最小之物品依序取物，直到物品拿完或負重 = W為止，就可以得到一個可行解
    - 利潤與重量比: 採用最大利潤與重量比的選擇程序。自利潤與重量比最大之物品依序取物，直到物品拿完或負重 = W為止，就可以得到一個可行解
  - 以上三種選擇程序，只有利潤與重量比可以得到一個最佳解，其餘兩個只能得到可行解
  - 因此，貪婪法則的選擇程序適題與否，對於是否可以得到一個問題之最佳解具有決定性的影響

- 根據題目定義，我們可以得到下列表格:

Item	重量 (bl)	利潤	利潤/重量比
1	5	$50	10
2	10	$60	6
3	20	$140	7

- 選擇程序採 "最大利潤優先":
  - **Step 1:** 取 20 bl的Item 1，可得利潤為 $140，背包剩餘重量: 10 bl
  - **Step 2:** 取10 bl的Item 2，連同Step 1所取之20 bl的Item 1，可得總利潤為 $200，背包剩餘重量: 0 bl
  - **Step 3:** 因為背包已無剩餘重量，故完全無法取得Item 3
  - **所得總利潤 = $200**

- 選擇程序採 "最小重量優先":
  - Step 1: 取 5 bl的Item 1，可得利潤為 $50，背包剩餘重量: 25 bl
  - Step 2: 取10 bl的Item 2，連同Step 1所取之5 bl的Item 1，可得總利潤為 $110，背包剩餘重量: 15 bl
  - Step 3: 由於背包剩餘重量為15 bl，而Item 3的重量有20 bl，因此僅能取 ¾ 的 Item 3，連同前兩步的結果，可得總利潤為 $215，背包剩餘重量: 0 bl
  - 所得總利潤 = $215
- 選擇程序採 "最大利潤與重量比":
  - Step 1: 取 5 bl的Item 1，可得利潤為 $50，背包剩餘重量: 25 bl
  - Step 2: 取20 bl的Item 3，連同Step 1的結果，可得總利潤為 $190，背包剩餘重量: 5 bl
  - Step 3: 由於背包剩餘重量為5 bl，而Item 2的重量有10 bl，因此僅能取 ½ 的 Item 2，連同前兩步的結果，可得總利潤為 $220，背包剩餘重量: 0 bl
  - 所得總利潤 = $220

Input: 物品數 **n;** 價值矩陣 **v[];** 重量矩陣 **w[];** 背包最大負重 **W**
Output: 在背包中的物品矩陣 **x[]**

```
procedure Fractional_Knapsack(n, v[], w[], W)
{
    x[] = ø;
    weight = 0;
    按照 v/w 排序 n 件物品;
    while (weight ≤ W)
    {
        k ← 剩餘物品中，v/w 最大者;
        if (weight + w[k] < W)
            將 k 放入背包中(即：x[k] ← 1)，且 weight ← weight + w[k];
        else
            discard k;
        if (weight == W 或 沒有物品可裝)
            break;
    };
}
```

時間複雜度分析:

- **排序需要O(n log n)**
- **while迴圈最多不會超過O(n)**

∴整個演算法為 O(n log n)

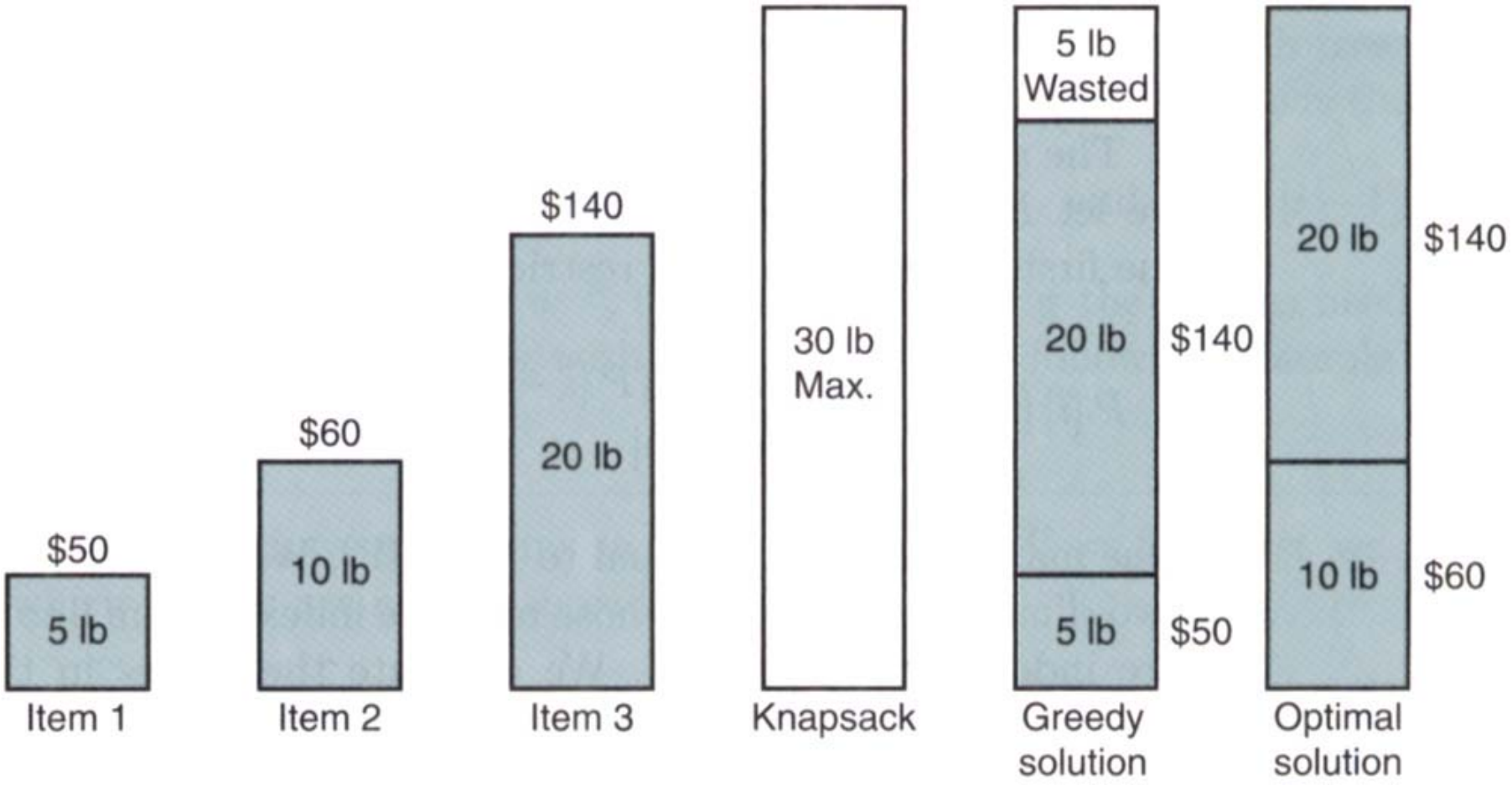
※練習範例※

- Now, you are inside a Buffet restaurant. Assume that your stomach can only accept food with maximum size M and there are n kinds of food with sizes and values as (s1, v1), (s2, v2), …, (sn, vn), in the restaurant. Design an algorithm to find the highest value of food that you can eat. Only need to demonstrate your algorithm based on the case: M = 22 and 4 kinds of food with (3, 4), (4, 5), (6, 9), (8, 13).
  - Hint: this is equivalent to the knapsack problem. (91年交大資工所)

Ans: 取食物的順序 : 4, 3, 1, 2, 其餘部份請自行說明。

0/1 Knapsack Problem

- 物品不可被切割，亦即取物時得取全部
- 若仍採用Greedy Approach，選擇程序為 "最大利潤與重量比":
  - Step 1: 取 5 bl的Item 1，可得利潤為 $50，背包剩餘重量: 25 bl
  - Step 2: 取20 bl的Item 3，連同Step 1的結果，可得總利潤為 $190，背包剩餘重量: 5 bl
  - Step 3: 由於背包剩餘重量為5 bl，而Item 2的重量有10 bl。由於物品不可被分割，因此完全無法取得Item 2，而背包剩餘重量: 5 bl
  - 所得總利潤 = $190，但是真正的最佳解 (最佳總利潤) 為 200
    - ∴ 0/1 Knapsack Problem不可用Greedy Approach求解!!


$50
5 lb
Item 1
$60
10 lb
Item 2
$140
20 lb
Item 3
30 lb
Max.
Knapsack
5 lb
Wasted
20 lb
$140
5 lb
$50
Greedy
solution
20 lb
$140
10 lb
$60
Optimal
solution

- 0/1背包問題所會使用到的2個資料結構：
  - **P[i, k]**
    - 背包可負重k且有 i 樣物品$\{O_1, O_2, ..., O_i\}$ 可拿之下，所能得到之最高利潤
    - i: 可以拿的物品數，k: 包包所能夠承載的重量
  - **label[i, k]**
    - 背包可負重k且有 i 樣物品$\{O_1, O_2, ..., O_i\}$ 可拿之下，所能得到之取物方法
- **[0/1背包問題的遞迴設計概念]:**
  - 給一背包可負重W，且可拿的物品 $O = \{O_1, O_2, ..., O_n\}$ 共n項，其中O_i的重量為w_i，利潤為p_i，取物時得全取。設 $\{x_1, x_2, ..., x_j\} \leq O$ 為具有最高獲利之物品取法，此時:
    - 若 $x_j = O_n$，則 $\{x_1, x_2, ..., x_{j-1}\}$ 為 "前n-1項物品$\{O_1, O_2, ..., O_{n-1}\}$之最佳取法，且該取法之負重為 $W - w_n$"
    - 若 $x_j \neq O_n$，則 $\{x_1, x_2, ..., x_j\}$ 為 "前n-1項物品$\{O_1, O_2, ..., O_{n-1}\}$之最佳取法，且該取法之負重為W"

- 課本上的解釋 (P. 4-53的4.5.3節)：
  - 為了達到利用Dynamic Programming解決0/1背包問題的目標，令A為n個物品的最佳子集合，則會有下列兩種情況：
    - **A包含$item_n$**：若A包含$item_n$，則A中物品的總利潤等於p_n加上由首n-1項物品中進行挑選所得到之最佳利潤，且挑選時遵守重量不能達到$W-w_n$的限制。
    - **A不包含$item_n$**：若A不包含$item_n$，則A等於首n-1項物品之最佳化子集合。

- [遞迴式]:

⑤ 第 i 物的重量比背包目前可承受之重量還重

① 沒物品可拿

② 無法負重

$$P[i,k]=\begin{cases}0 & \text{if } i=0 \text{ or } k=0\\ P[i-1,k] & \text{if } k<w_i\\ \max(p_i+P[i-1,k-w_i],P[i-1,k]) & \text{if } k\geq w_i\end{cases}$$

③ 拿第 i 物後可得的利潤

④ 不拿第 i 物可得的利潤

- 範例: 假設有一背包W = 5，考慮以下的Items，求0/1 Knapsack最佳解:

Item	重量	利潤
O_1	1	$6
O_2	2	$10
O_3	3	$12

Sol:

- 先建立P[0...n, 0...W]和label[0...n, 0...W]兩個Table

P	0	1	2	3	4	5
0						
1						
2						
3						

label	0	1	2	3	4	5
0						
1						
2						
3						

Item	重量	利潤
O_1	1	$6
O_2	2	$10
O_3	3	$12

$$P[i,k] = \begin{cases} 0 & \text{if } i = 0 \text{ or } k = 0 \\ P[i-1,k] & \text{if } k < w_i \\ \max(p_i + P[i-1, k-w_i], P[i-1,k]) & \text{if } k \geq w_i \end{cases}$$

- **Step 1:** 當 i = 0，表示沒有任何物品可以拿 (即: 狀況 ①)。

  ∴ P[0, k] = 0 且 label[0, k] = {ø}

P	0	1	2	3	4	5
0						
1						
2						
3						

label	0	1	2	3	4	5
0						
1						
2						
3						

Item	重量	利潤
O_1	1	$6
O_2	2	$10
O_3	3	$12

$$P[i,k]=\begin{cases}0 & \text{if } i=0 \text{ or } k=0\\ P[i-1,k] & \text{if } k<w_i\\ \max(p_i+P[i-1,k-w_i],P[i-1,k]) & \text{if } k\geq w_i\end{cases}$$

- **Step 2:** 當 **i = 1**，表示有 **1** 個物品可以拿 **(即: O_1)**。
  - ∵**k = 0** ，表示無法負重 (狀況 ②); ∴**P[1, 0] = 0** 且 **label[1, 0] = ø**
  - **k=1**，表示能負重**1;** 此時:

$$P[1,1]=\max\begin{cases}P[0,1]\\ p_1+P[0,0]\end{cases}=\max\begin{cases}0\\ 6+0\end{cases}=6$$

  - **K = 2~5**，表示能負重**2 ~ 5;** 但由於僅**1**個物品 **(即: O_1)** 可拿，因此 **P[1, k]**與**label[1, k]**的其它值皆同**P[1, 1]**與**label[1, 1]**。

P	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						

label	0	1	2	3	4	5
0	ø	ø	ø	ø	ø	ø
1						
2						
3						

Item	重量	利潤
O_1	1	$6
O_2	2	$10
O_3	3	$12

$$P[i,k]=\begin{cases}0 & \text{if } i=0 \text{ or } k=0\\ P[i-1,k] & \text{if } k<w_i\\ \max(p_i+P[i-1,k-w_i],P[i-1,k]) & \text{if } k\geq w_i\end{cases}$$

- **Step 3:** 當 **i = 2**，表示有 **2** 個物品可以拿 **(**即: O_1 與 O_2**)**。
  - ∵**k = 2** ，表示負重**2;** 此時:

$$P[2,2]=\max\begin{cases}P[1,2]\\ p_2+P[1,0]\end{cases}=\max\begin{cases}6\\ 10+0\end{cases}=10$$

  - ∵**k = 3** ，表示負重**3;** 此時:

$$P[2,3]=\max\begin{cases}P[1,3]\\ p_2+P[1,1]\end{cases}=\max\begin{cases}6\\ 10+6\end{cases}=16$$

P	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	6	6	6	6	6
2						
3						

label	0	1	2	3	4	5
0	ø	ø	ø	ø	ø	ø
1	ø	{1}	{1}	{1}	{1}	{1}
2						
3						

Item	重量	利潤
O_1	1	$6
O_2	2	$10
O_3	3	$12

$$P[i,k]=\begin{cases}0 & \text{if } i=0 \text{ or } k=0\\ P[i-1,k] & \text{if } k<w_i\\ \max(p_i+P[i-1,k-w_i],P[i-1,k]) & \text{if } k\geq w_i\end{cases}$$

- **Step 4:** 當 **i = 3**，表示有 **3** 個物品可以拿 (即: O_1、O_2與 O_3)。
  - ∵**k = 3** ，表示負重**3;** 此時:

$$P[3,3]=\max\begin{cases}P[2,3]\\ p_3+P[2,0]\end{cases}=\max\begin{cases}16\\ 12+0\end{cases}=16$$

P	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	6	6	6	6	6
2	0	6	10	16	16	16
3						

label	0	1	2	3	4	5
0	ø	ø	ø	ø	ø	ø
1	ø	{1}	{1}	{1}	{1}	{1}
2	ø	{1}	{2}	{1, 2}	{1, 2}	{1, 2}
3						

Item	重量	利潤
O_1	1	$6
O_2	2	$10
O_3	3	$12

$$P[i,k]=\begin{cases}0 & \text{if } i=0 \text{ or } k=0\\ P[i-1,k] & \text{if } k<w_i\\ \max(p_i+P[i-1,k-w_i],P[i-1,k]) & \text{if } k\geq w_i\end{cases}$$

- ∵ **k = 4** ，表示負重**4;** 此時:

$$P[3,4]=\max\begin{cases}P[2,4]\\ p_3+P[2,1]\end{cases}=\max\begin{cases}16\\ 12+6\end{cases}=18$$

- ∵ **k = 5** ，表示負重**5;** 此時:

$$P[3,5]=\max\begin{cases}P[2,5]\\ p_3+P[2,2]\end{cases}=\max\begin{cases}16\\ 12+10\end{cases}=22$$

P	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	6	6	6	6	6
2	0	6	10	16	16	16
3	0	6	10	16		

label	0	1	2	3	4	5
0	ø	ø	ø	ø	ø	ø
1	ø	{1}	{1}	{1}	{1}	{1}
2	ø	{1}	{2}	{1, 2}	{1, 2}	{1, 2}
3	ø	{1}	{2}	{1, 2}		

【0/1 Knapsack 演算法】

Input：物品數：n，價值矩陣：$v[\,]$，重量矩陣：$w[\,]$，背包最大負重：W

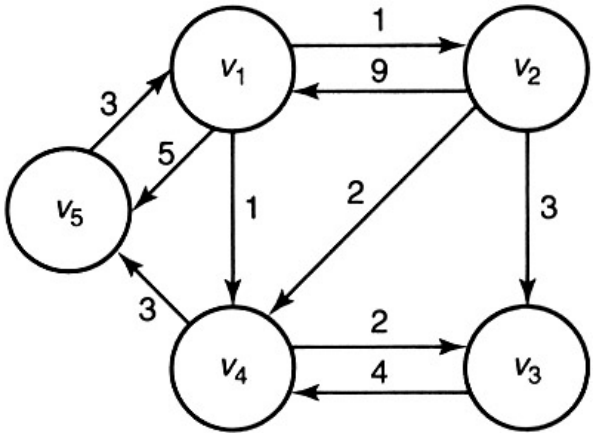
Output：拿取之物品 $label[n, W]$及其價值 $c[n, W]$

```
1   for k ← 0 to W
2       c[0, k] ← 0;
3       label[0, k] ← φ;
4   for i ← 1 to n
5       c[i, 0] ← 0;
6       label[i, 0] ← φ;
7       for k ← 1 to W
8           if w[i] ≦ k
9               then if v[i]+c[i−1, k−w[i]] > c[i−1, k]
10                  then c[i, k] ← v[i]+c[i−1, k−w[i]];
11                      label[i, k] ← label[i−1, k−w[i]] ∪ {i};
12              else c[i, k] ← c[i−1, k];
13                  label[i, k] ← label[i−1, k];
14      else c[i, k] ← c[i−1, k];
15          label[i, k] ← label[i−1, k];
```

① ② ③ ④ ⑤

Dijkstra Algorithm for Single-pair Shortest Path Problem

- 最短路徑 (Shortest Path) 問題
  - 求單一頂點到其它頂點之最短路徑 (Single pair shortest path)
    - 使用Dijkstra's Algorithm
      - 採用 "貪婪演算法" 之解題策略
      - 找出某一頂點到其它頂點之最短路徑之時間複雜度為$O(n^2)$
  - 求所有頂點之間的最短路徑 (All pair shortest path)
    - 使用n次Dijkstra's Algorithm
      - 每一次帶不同的起始點
      - 需要的時間複雜度 $O(n^3)$
    - 使用Floyd's Algorithm
      - 採用 "動態規劃" 之解題策略



- Dijkstra's Algorithm有三個資料結構來做輔助：
  - **Cost Matrix** (成本矩陣, 相鄰矩陣):
    - 假設G = (V, E)為一有向圖，邊上有路徑長度 (or 成本)，|V|=n，則Cost Matrix為一個nxn的二維矩陣，其中：

$$Cost[i, j] = \begin{cases} Cost(i, j), & \text{if } < vi, vj > \in E \\ 0, & \text{if } vi = vj \\ \infty, & \text{if } < vi, vj > \notin E \end{cases}$$

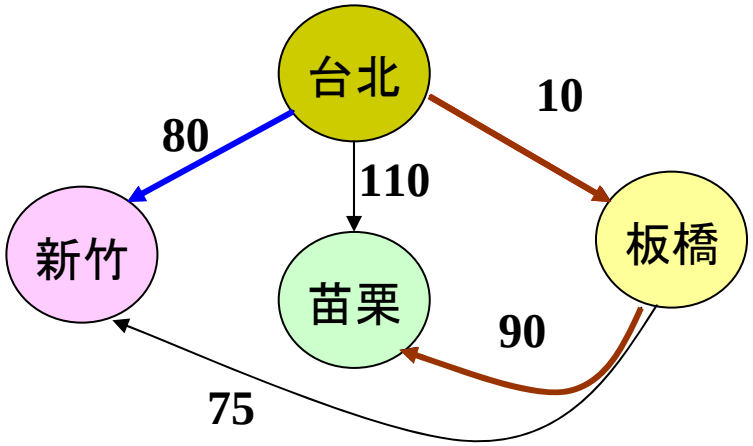
  - **S[1... n]** of Boolean:
    - 初始值皆為 0

$$S[i] = \begin{cases} 0, & \text{if 起點到頂點 vi之最短路徑未決定} \\ 1, & \text{if 起點到頂點 vi之最短路徑已確定} \end{cases}$$

  - **DIST[1... n]** of Integer:
    - DIST[i] 表示由目前起點到頂點Vi之最短路徑長度 (Shortest Path Length)
    - 此array存放我們最終所需要的結果

Dijkstra的解題概念

- 從某一起始點v到其它頂點的最短路徑 (起始點v自己到達自己一定是最短，且路徑長度為0)：
  1. 選擇程序：排除已記錄到S矩陣中的頂點，找出從起始點v到達哪一個終止頂點vj的路徑最短 (此路徑不排除會經過S矩陣中所記錄的各個頂點)，記下該頂點vj (by S[j])，同時將起始點至該點的最短路徑長度記錄到DIST[j]。
     - 若S矩陣內的值皆為0，則為初始情況，尚未知有任何頂點vi與v有最短路徑。
     - 此時主要考量從起始點v到達哪一個終止頂點vi的直達路徑最短，由S矩陣記錄該頂點vi ，同時將起始點至該點的最短路徑長度記錄到DIST[i]。
  2. 重覆步驟1，直到圖中所有頂點皆被記錄到S矩陣為止。

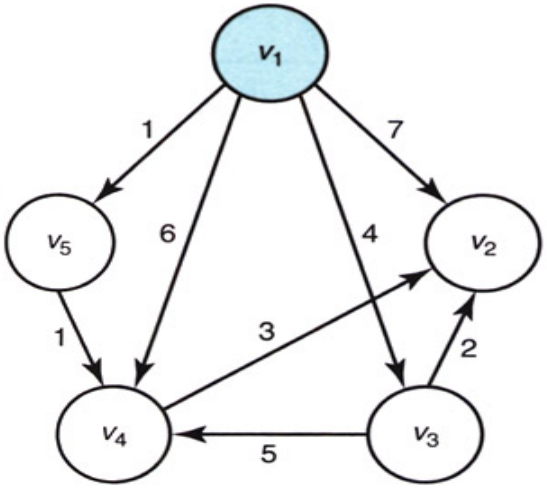


S[ ]

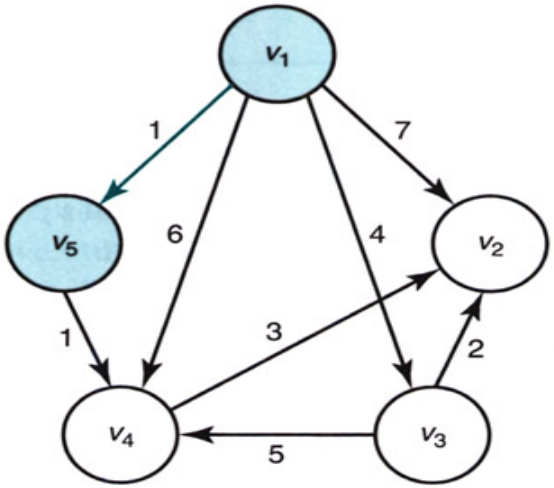
DIST[ ]

Cost	台北	板橋	新竹	苗栗
台北	0	10	80	110
板橋	∞	0	75	90
新竹	∞	∞	0	∞
苗栗	∞	∞	∞	0

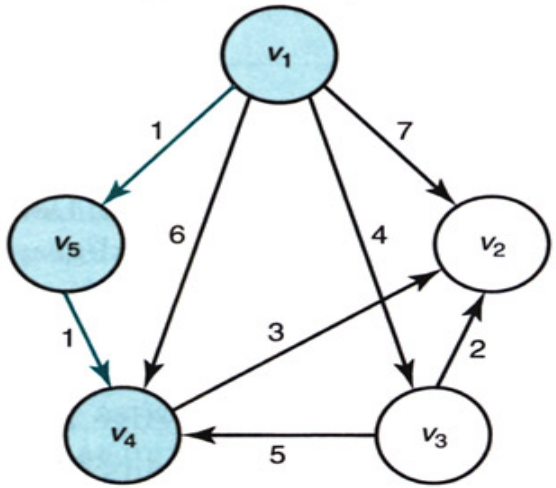
Compute shortest paths from v_1.



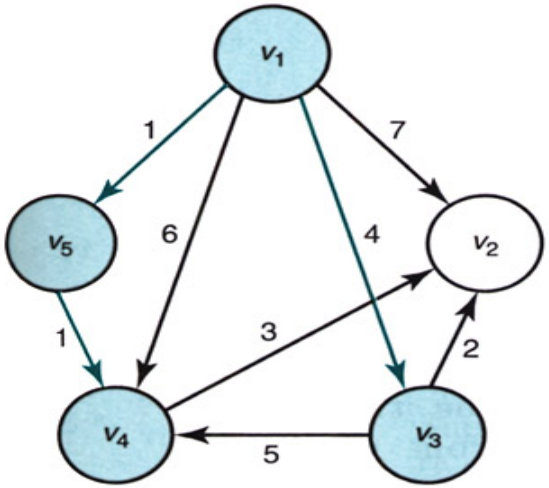
1. Vertex v_5 is selected because it is nearest to v_1.



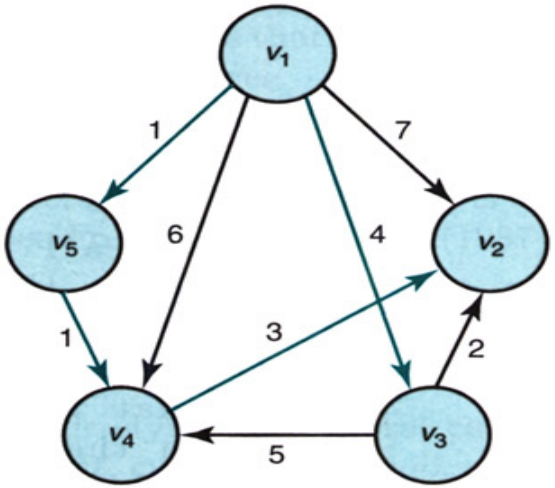
2. Vertex v_4 is selected because it has the shortest path from v_1 using only vertices in $\{v_5\}$ as intermediates.



3. Vertex v_3 is selected because it has the shortest path from v_1 using only vertices in $\{v_4, v_5\}$ as intermediates.

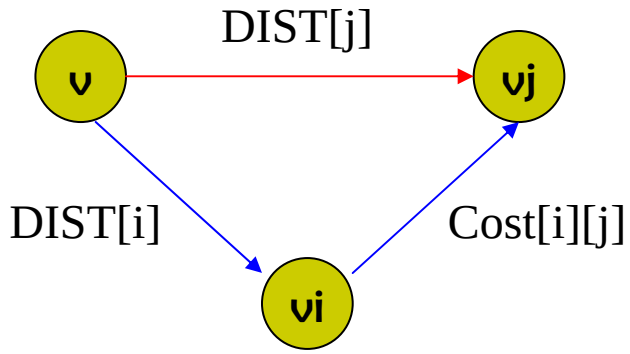


4. The shortest path from v_1 to v_2 is $[v_1, v_5, v_4, v_2]$.



Dijkstra的解題程序

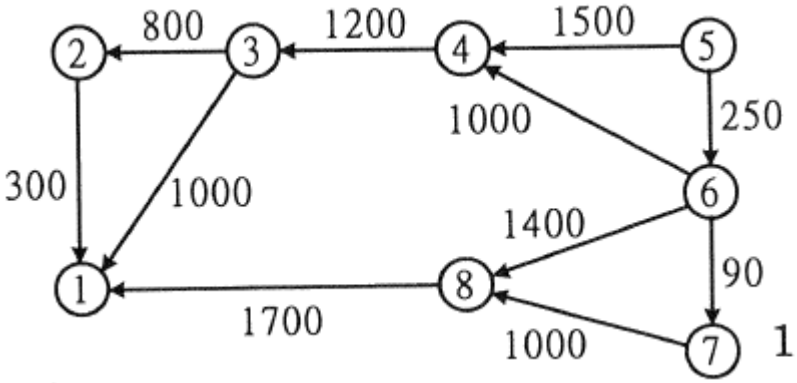
1. DIST[ ]的初值為Cost矩陣第v列 (令 v 為起點)
   - S[1 ... n] = 0
   - S[v] = 1
2. 自那些尚未決定shortest path的頂點 (即：S[]中仍為0之頂點) 中，挑出最小DIST[i]。
3. S[i] = 1 (表示已確定起點v到頂點vi的最短路徑)
4. 更新DIST[j] (for S[j] = 0)by pass through vi



$$DIST[j] = \min(DIST[j], DIST[i] + Cost[i, j])$$

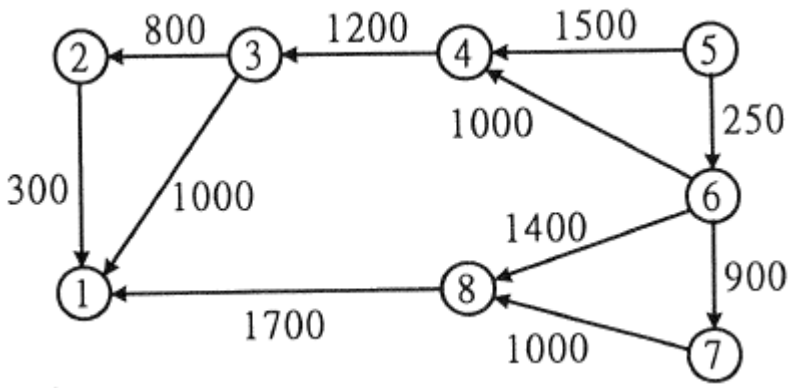
5. Repeat 2 ~4 until all S[i] = 1

- 下圖是一個含有8個頂點的有向圖，其成本矩陣如下所示：



	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	1000	800	0					
4			1200	0				
5				1500	0	250		
6				1000		0	900	1400
7							0	1000
8	1700							0

相鄰矩陣，未表示的值為∞



	1	2	3	4	5	6	7	8
S	0	0	0	0	0	0	0	0

狀態	頂點選擇 u	DIST	1	2	3	4	5	6	7	8
Initial										
1										
2										
3										
4										
5										
6										
7										

```
Dijstra's Algorithm (v, Cost[][], DIST[], n)
{
  Boolean S[1...n];
  int num;
  for (i=1; i≤n; i++)
    {
     S[i] = 0;
     DIST[i] = Cost[v][i];
    };
  S[v] = 1;
  DIST[v] = 0;
  num = 2;
  while (num < n)
    {
     for (all w withS[w] = 0)
        Choose u，此 u 滿足: DIST[u] = min{DIST[w] | S[w] = 0};
     S[u] = 1;
     num++;
     for (all w withS[w] = 0)
        DIST[w] = min{DIST[w], DIST[u]+Cost[u][w]};
    };
}
```

初值設定。其中：

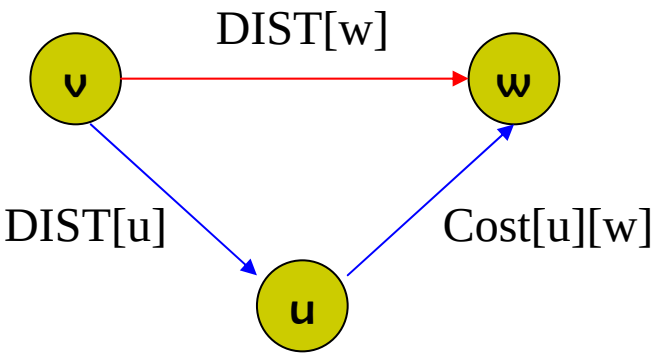
- for迴圈
  - 將一維矩陣S[ ]全設成0
  - 將一維矩陣DIST[ ]設成與二維矩陣Cost第v列之所有值相同!!
- S[v] = 1是將頂點v自己到自己的最短路徑設為 "已確定"。
- DIST[v] = 0是將頂點v自己到自己的最短路徑的距離設為 0

此for迴圈主要是從一維矩陣DIST[ ]，**找出**尚未決定最短路徑的頂點中，符合最短路徑之頂點u

此for迴圈主要是**更新**一維矩陣DIST[ ]中，尚未決定最短路徑的頂點w之最短路徑值

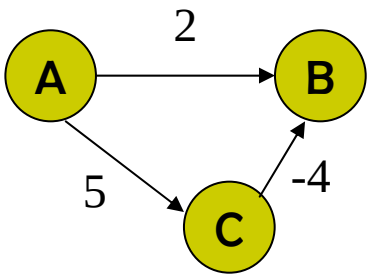
Time Complexity: $O(n^2)$

- Dijkstra's Algorithm觀念圖解:



$$DIST[w] = \min(DIST[w], DIST[u] + Cost[u, w])$$

- **Dijkstra's Algorithm**成立的假設條件：
  - 圖形中不能存在有**Negative Cost** 的邊，否則可能無法正常運作。
  - 例：



上圖若依**Dijkstra's Algorithm**，會求出**DIST[B] = 2**。然而，**A→B**最短路徑為 **1**。

這是因為：

- **S[B] = 0 → 1, DIST[B] = 2**
- **S[C] = 0 → 1, DIST[C] = 5**
- ∵**S[B]**已等於 1，∴無法透過**C**來更新**DIST[B]**

補　充

再探Input Size (輸入大小)

- 一般的演算法複雜度分析當中，我們通常將 "輸入到演算法中的資料量" 視為輸入大小 **(Input Size)**。
- **When we are determining whether an algorithm is polynomial-time, it is necessary to be careful about what we call the input size.**
  - 如果要求算第n個費氏數，我們可能會誤認 n 就是Input Size
  - 如: n = 13，只是計算費氏數程式的輸入資料量
  - 合理的輸入大小應該是用來對n個資料進行編碼的符號數目
    - 如果使用二進位法，輸入大小將是用了多少個位元對 n 進行編碼，也就是⌊lg n⌋ + 1。例如:

$$n = 13 = \underbrace{1101}_{4 \text{ bits}}{}_2$$

- **Ex1:** 以下列有 n 個數字相加之演算法說明:

```
int i, x;
x = 0;
for (i=1; i<=n; i++)
    x = x + i;
```

  - 以上例來說，n 是輸入的數量，lg n 才是輸入的大小
  - 若 n = 10，表示會有10個整數做相加(數量)，而每一個輸入值的大小是⌊lg 10⌋+1 = 4 bits (∵ 4bits 可表示的輸入值變化為 0 ~ 15)
  - 因此，以輸入大小來看，此演算法的時間複雜度為指數 $(2^{\lg n})$

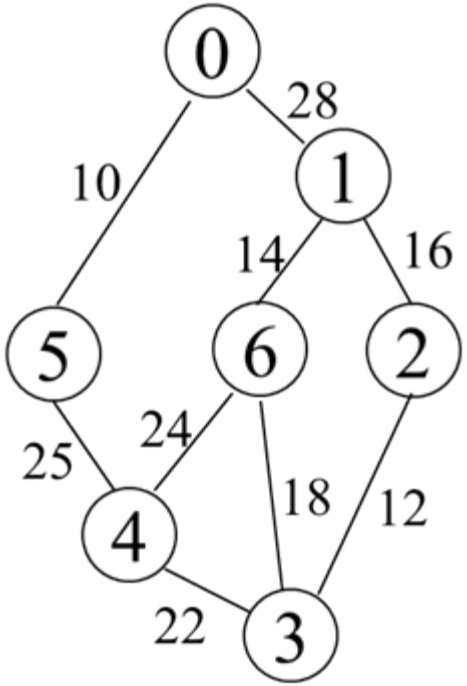
- **Ex:** 分析0/1背包問題:
  - **Time Complexity:** $\Theta(nW)$. //n: 可拿物品的個數; W: 背包可承受之重量
  - **Space Complexity:** $\Theta(nW)$. //宣告了多大的Array來儲存
  - n 為項目數，W為數量
    - lg W為背包重量的輸入大小
  - 因此， 0/1背包問題的演算法:
    - 以資料量而言，其時間複雜度是多項式，即: $\Theta(nW)$
    - 若以輸入大小而言，其時間複雜度則是指數成長，即: $\Theta(n\,2^{\lg W})$
- **Pseudo polynomial-time (虛擬多項式時間)**
  - 若某個演算法被稱為是虛擬多項式時間的演算法，通常只有在碰到含有極大數值的輸入資料狀況，這類的演算法才會沒有效率。
  - For example, in the 0–1 Knapsack problem, we might often be interested in cases where *W* is not extremely large.

Sollin's Algorithm

- **Sollin's Algorithm 解題要件**:
  - **選擇程序**:以**Tree Edge**為設計概念。在執行過程中，為每個樹選擇一個最小成本的**Tree Edge**。
  - **限制條件**:不允許有迴路
- **Steps:**
  1. 一開始，每個頂點皆視為獨立的**Tree Root**。
  2. 從各樹中，挑選出具最小成本的樹邊 。
  3. 刪除重覆挑選的樹邊，僅保留一個即可。
  4. **Repeat** ②~③直到下列任一條件成立為止:
     - 只剩一個**Tree**
     - 無邊可挑
  5. 若 **|T| < n-1**，則無**Spanning Tree**。**(|T|**:樹的邊數**)**

範例 1

- 試利用**Sollin's Algo.**求下圖的**Minimum Spanning Tree**



Sol: (第一輪)

① 一開始，每個頂點皆視為獨立的**Tree Root**。

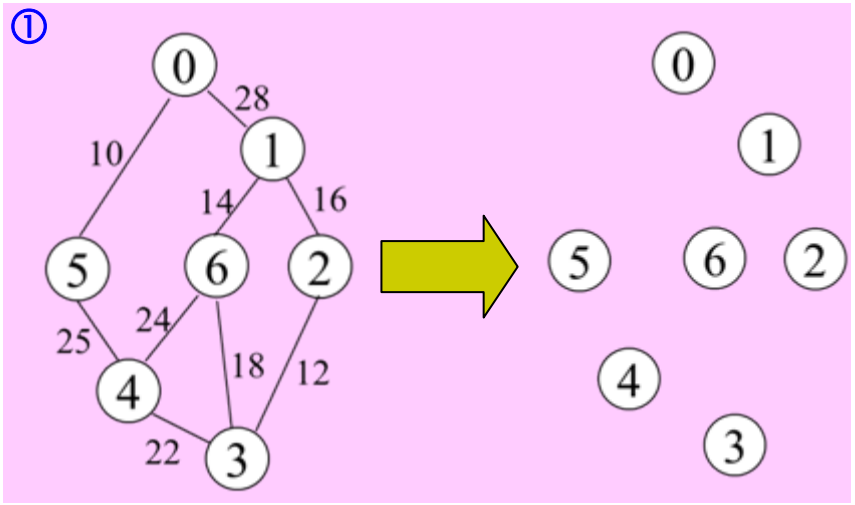
② 從各樹中，挑選出具最小成本的樹邊 。

③ 刪除重覆挑選的樹邊，僅保留一個即可。

④ **Repeat** ②~③直到下列任一條件成立為止:

- 只剩一個**Tree**
- 無邊可挑

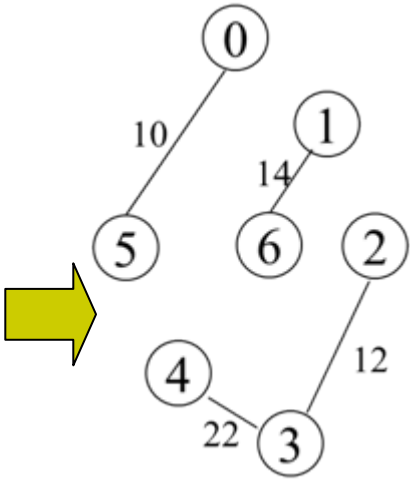
⑤ 若 **|T| < n-1**，則無**Spanning Tree**。**(|T|**：樹的邊數**)**



②

(0, 5)
(1, 6)
(2, 3)
~~(3, 2)~~ 與(2, 3)重覆
(4, 3)
~~(5, 0)~~ 與(0, 5)重覆
~~(6, 1)~~ 與(1, 6)重覆

③



尚有三棵Tree，故須執行第二輪

Sol: (第二輪)

~~① 開始，每個頂點皆視為獨立的Tree Root。~~

② 從各樹中，挑選出具最小成本的樹邊 。

③ 刪除重覆挑選的樹邊，僅保留一個即可。

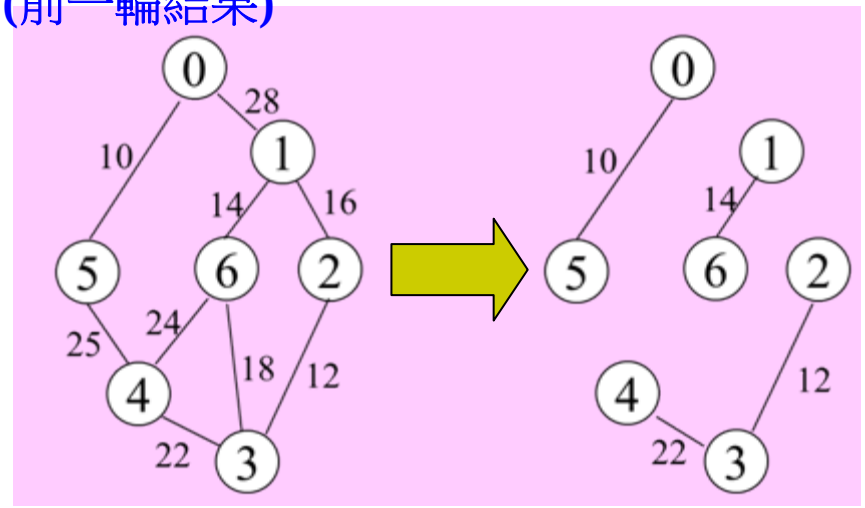
④ **Repeat** ②~③直到下列任一條件成立為止:

- 只剩一個**Tree**
- 無邊可挑

⑤ 若 **|T| < n-1**，則無**Spanning Tree**。(**|T|**：樹的邊數)

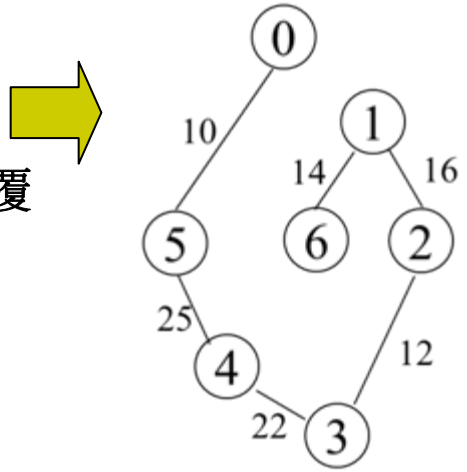
僅剩一棵Tree，故停止Tree的產生

(前一輪結果)



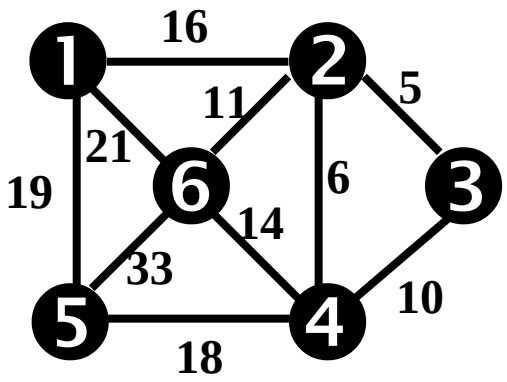
② (5, 4)
(1, 2)
~~(2, 1)~~ 與(1, 2)重覆

③



範例 2

- 試利用**Sollin's Algo.**求下圖的**Minimum Spanning Tree**



① 一開始，每個頂點皆視為獨立的**Tree Root**。

② 從各樹中，挑選出<u>具最小成本的樹邊</u> 。

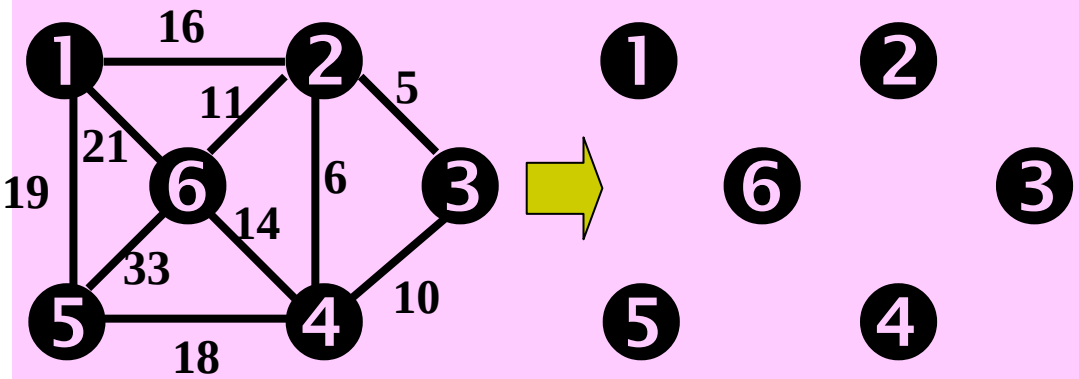
③ 刪除重覆挑選的樹邊，僅保留一個即可。

④ **Repeat** ②~③直到下列任一條件成立為止：

- 只剩一個**Tree**
- 無邊可挑

⑤ 若 **|T| < n-1**，則無**Spanning Tree**。**(|T|**：樹的邊數)

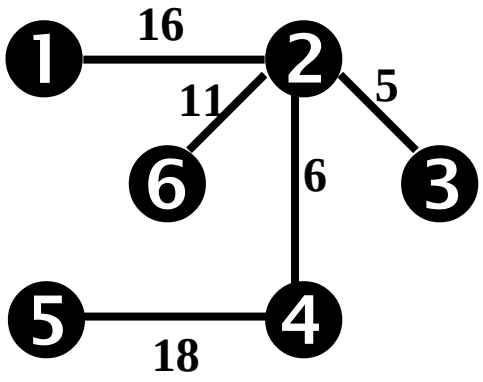
①



②

(1, 2)
(2, 3)
~~(3, 2)~~ 與(2, 3)重覆
(4, 2)
(5, 4)
(6, 2)

③



僅剩一棵Tree，故停止Tree的產生

Summary

- 時間複雜度為$O(n^2)$，其中$n = |V|$
- 與**Kruskal's Algo.**及**Prime's Algo.**一樣，皆屬於 "**Greedy**" 策略
- 與**Kruskal's Algo.**及**Prime's Algo.** 兩個演算法比較起來，仍是以**Kruskal's algo.** 較為快速 (∵ **n log n**)
- 其它**Summary**請見**Slide 21**的整理。